DEVELOPER GUIDE

REST API Fundamentals

A practical introduction for developers

- HTTP methods and status codes
- Authentication patterns
- Rate limiting, pagination and error handling

10 pages | Reference material for backend and integration work

What Is a REST API?

The architectural style behind most modern web services

Resources, not actions

Every entity is addressed by a URL. /users/42 identifies one user; the HTTP method decides what happens to it.

Stateless requests

The server keeps no session memory between calls. Each request carries everything needed to process it, including credentials.

Uniform interface

The same small set of HTTP methods works across every resource, so a new endpoint is predictable once you know the pattern.

Representations

A resource is returned in a format - almost always JSON today, though XML and others are valid.

HTTP Methods

What each verb means, and whether repeating it is safe

METHOD	PURPOSE	TYPICAL SUCCESS	IDEMPOTENT	SAFE
GET	Retrieve a resource	`200 OK`	Yes	Yes
POST	Create a new resource	`201 Created`	No	No
PUT	Replace a resource fully	`200 / 204`	Yes	No
PATCH	Update part of a resource	`200 OK`	No	No
DELETE	Remove a resource	`204 No Content`	Yes	No

Why idempotency matters

An idempotent call can be retried after a timeout without side effects. Repeating POST may create duplicates, which is why many APIs accept an Idempotency-Key header on POST.

Status Codes You Will Actually See

The first digit tells you who is responsible

2xx Success

- **200** OK - request succeeded
- **201** Created - new resource made
- **202** Accepted - queued, not done yet
- **204** No Content - success, empty body

4xx Client error

- **400** Bad Request - malformed input
- **401** Unauthorized - missing/bad credentials
- **403** Forbidden - authenticated, not allowed
- **404** Not Found - no such resource
- **409** Conflict - state clash
- **422** Unprocessable - validation failed
- **429** Too Many Requests - rate limited

5xx Server error

- **500** Internal Server Error
- **502** Bad Gateway - upstream failed
- **503** Service Unavailable - overloaded

Rule of thumb: 4xx means fix your request; 5xx means retry later with backoff.

Anatomy of a Request

Method, path, headers, body - then the response

Request

```
POST /v1/articles HTTP/1.1
Host: api.example.com
Authorization: Bearer <token>
Content-Type: application/json

{
  "title": "My article",
  "status": "published"
}
```

Response

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /v1/articles/8821

{
  "id": 8821,
  "title": "My article",
  "url": "https://.../8821"
}
```

Headers worth setting every time

```
Content-Type       tells the server how to parse your body
Accept             tells the server what format you want back
Authorization      carries your credentials
User-Agent         identifies your client in server logs
```

Authentication Patterns

Four common approaches, from simplest to most involved

- **API key**
  A long static string sent in a header. Easy to start with, but it never expires on its own - so rotation and storage discipline matter.

```
X-API-Key: abc123...
```

- **Bearer token**
  A token issued after login, sent on every call. Usually short-lived, which limits the damage if one leaks.

```
Authorization: Bearer eyJhbGci...
```

- **Basic auth**
  Username and password, base64 encoded. Only acceptable over HTTPS, since base64 is encoding, not encryption.

```
Authorization: Basic dXNlcjpwYXNz
```

- **OAuth 2.0**
  Delegated access without sharing a password. You receive an access token plus a refresh token to obtain new ones.

```
grant_type=refresh_token&...
```

Rate Limiting and Retries

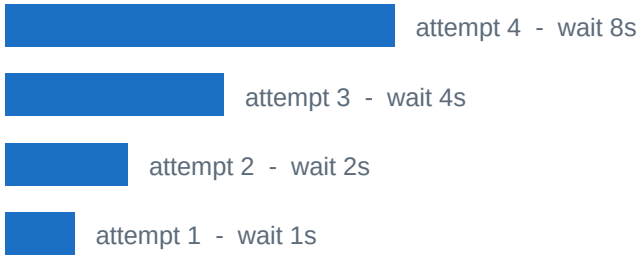
Every serious API throttles you - plan for it

Headers that tell you where you stand

```
HTTP/1.1 429 Too Many Requests
Retry-After: 30
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1712345678
```

Exponential backoff

Wait longer after each failure instead of hammering the endpoint.



Practical rules

- Respect Retry-After when the server sends it - guessing is worse than reading.
- Add jitter (a small random delay) so parallel clients do not retry in lockstep.
- Cap the number of retries; failing loudly beats retrying forever.
- Never retry a 4xx other than 429 - the request itself is the problem.

Pagination

Three strategies, with different trade-offs

Offset / limit

`?limit=50&offset=100`

Simple to implement and reason about.

Rows shift if data changes mid-scan,

so items can repeat or be skipped.

Gets slow on very large offsets.

Cursor based

`?limit=50&after=eyJpZCI6MTAwfQ`

Stable even while data changes.

Efficient at any depth.

Cannot jump to an arbitrary page.

Preferred by most large APIs.

Page number

`?page=3&per_page=50`

Friendliest for building UI controls.

Same drift problem as offset.

Often paired with a total count.

Fine for small, stable datasets.

If you are designing the API: default to cursor pagination and always send a stable sort order.
If you are consuming one: follow the next link the response gives you rather than building URLs yourself.

Designing Useful Errors

A good error message saves hours of debugging

Unhelpful

```
HTTP/1.1 400 Bad Request

{
  "error": true,
  "message": "Invalid request"
}
```

Helpful

```
HTTP/1.1 422 Unprocessable Entity

{
  "code": "validation_failed",
  "message": "Title is too long.",
  "field": "title",
  "max_length": 120,
  "docs": "https://.../errors"
}
```

What to include

- A stable machine-readable code, so clients can branch on it without parsing prose.
- A human-readable message that names the actual problem.
- The offending field, when the error is a validation failure.
- A link to documentation - and never leak stack traces or internal paths.

Practical Checklist

Worth reviewing before you ship an integration

- [ ] Always use HTTPS - never send tokens over plain HTTP.
- [ ] Keep credentials in environment variables, not in source code.
- [ ] Version the API in the path, for example /v1/.
- [ ] Send Content-Type and Accept on every call.
- [ ] Handle 429 with backoff before you hit production traffic.
- [ ] Log request IDs from responses - support teams ask for them.
- [ ] Set explicit timeouts; a hung request is worse than a failed one.
- [ ] Validate input on the client too, to save a round trip.
- [ ] Follow next links rather than constructing pagination URLs.
- [ ] Test the unhappy paths: expired token, empty list, partial failure.

Further reading

MDN Web Docs - HTTP reference | RFC 9110 (HTTP Semantics) | RFC 6749 (OAuth 2.0)
Most API providers also publish a status page and a changelog - both are worth subscribing to.